Aligot: Cryptographic Function Identification in Obfuscated Binary Programs

Joan Calvet
Université de Lorraine, LORIA
Nancy, France
joan.calvet@loria.fr

José M. Fernandez
École Polytechnique
Montréal, Canada
jose.fernandez@polymtl.ca

Jean-Yves Marion
Université de Lorraine, LORIA
Nancy, France
jean-yves.marion@loria.fr

ABSTRACT

Analyzing cryptographic implementations has important applications, especially for malware analysis where they are an integral part both of the malware payload and the unpacking code that decrypts this payload. These implementations are often based on well-known cryptographic functions, whose description is publicly available. While potentially very useful for malware analysis, the identification of such cryptographic primitives is made difficult by the fact that they are usually obfuscated. Current state-of-the-art identification tools are ineffective due to the absence of easily identifiable static features in obfuscated code. However, these implementations still maintain the input-output (I/O) relationship of the original function. In this paper, we present a tool that leverages this fact to identify cryptographic functions in obfuscated programs, by retrieving their I/O parameters in an implementation-independent fashion, and comparing them with those of known cryptographic functions. In experimental evaluation, we successfully identified the cryptographic functions TEA, RC4, AES and MD5 in obfuscated programs. In addition, our tool was able to recognize basic operations done in asymmetric ciphers such as RSA.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security

Keywords

Binary Program Analysis, Malware, Cryptography

1. INTRODUCTION

Malicious software (malware) employs cryptography for many purposes, including hiding its network communications [6, 7, 26] and protecting its payload [5, 36, 22]. This use of cryptography is often combined with code obfuscation in order to thwart reverse engineering efforts by analysts and potentially avoid detection by security products. Obfuscation techniques in malware have become increasingly prevalent in the last few years, in particular due to the increased availability and use of obfuscating *code packers* by malware authors [21].

Analyzing and identifying the use of crypto functions is of a primary interest for several reasons. First, it can allow access to the decrypted malware core before it gets executed, which in turn provides key insight on the intentions of those creating and deploying the malware. Second, cryptographic code constitutes a feature that can be used to improve malware classification, an important step towards the identification of tools, methods and potentially groups or individuals behind particular malware attacks. Finally, an in-depth understanding of the decryption process and the identification of critical input parameters, such as the decryption key, can help in the development of *static unpackers* [27], i.e. automatic methods for extracting the core logic of packed programs without executing them. These tools allow analysts to write signatures on malware core features, often shared among variants, and can be easily deployed on end-user computers since they are lightweight.

As is the case with many binary program analysis tasks, cryptographic implementation reverse engineering is challenging, notably due to the complexity of the machine language and the lack of high-level structure. In practice, these implementations are often based on well-known cryptographic functions, whose description is publicly available. This opens the road for an alternative to cumbersome line-by-line analysis: the search for signs indicating which cryptographic function was implemented. Identifying this function does not directly provide cryptographic parameters, but it makes finding them much easier.

Most existing tools for cryptographic function identification in binary programs, like KANAL [33], DRACA [15] or Signsrch [4], are based on the recognition of code features by static analysis, such as specific constant values or machine language instructions, that are usually present in normal implementations of the recognized primitives. However, these tools are mostly ineffective on obfuscated programs, which have been made purposely hard to analyze and therefore hide static signs that could betray the presence of known cryptographic functions. Hence simple static-analysis identification of cryptographic functions is not suitable for such programs.

In contrast to this approach, we propose here a method

CCS'12, October 16–18, 2012, Raleigh, North Carolina, USA.

and a tool called *Aligot* for identifying cryptographic functions and retrieving their parameters, in a manner that is essentially independent of the actual implementation. Our solution leverages the particular input-output (I/O) relationship of cryptographic functions. Indeed if $\mathcal{F}_1$ is a cryptographic function such that $\mathcal{F}_1(K, C) = C'$, with K a key, C an encrypted text and C' a decrypted text, then it is very unlikely that another cryptographic function $\mathcal{F}_2$ verifies $\mathcal{F}_2(K, C) = C'$. In other words, the pair $((K, C), C')$ identifies $\mathcal{F}_1$ with overwhelming probability.

Consequently, if we observe during a particular execution of a program **P** that the values K and C are used to produce the value C', then we can conclude that **P** implements $\mathcal{F}_1$ during this particular execution. Of course, not all execution paths of **P** may implement $\mathcal{F}_1$, but identifying which potential paths are relevant is a separate reverse-engineering problem, albeit one that is handled with success by most skillful malware reverse engineers. Henceforth, we therefore restrict ourselves on the problem of identifying cryptographic functions along a given (chosen) execution path. In a nutshell, our method consists of (1) the retrieval of I/O parameters of possible cryptographic code during its execution, and (2) the comparison of the observed I/O relationship with those of known cryptographic functions.

We describe in §2 the previous work on cryptographic function analysis in binary programs. We provide an overview of our method in §3, and the detailed steps in §4 to 8. In §9 we report on our experimental evaluation of the method. We discuss the viability and limitations of our overall approach and conclude in §10 and 11.

2. RELATED WORK

The problem of analyzing cryptographic code and extracting its parameters in binary programs has been previously studied for different motivations. Within the context of Computer Forensics analysis, Halderman et al. [11] used particular properties of DES, AES and RSA to retrieve cryptographic parameters in the presence of bit-flipping errors. Maartmann-Moe *et al.* [19] extended this technique to the Serpent and Twofish ciphers. As these methods rely on such algorithm specific characteristics, they unfortunately require an in-depth study of each cipher.

As far as we know, Noe Lutz [18] was the first to explore generic cryptographic code detection using dynamic analysis. Lutz uses three indicators to recognize cryptographic code in execution traces: (1) presence of loops, (2) a high ratio of bitwise arithmetic instructions and (3) entropy change in the data manipulated by the code. In subsequent work, Wang *et al.* [32] and Caballero et al. [6] used similar observations to automatically retrieve decrypted data from encrypted communications. Several assumptions made in these works are not applicable to obfuscated programs. In particular arithmetic instructions are commonly used in junk code, making them an unreliable indicator of the presence of cryptographic code. Finally, none of these techniques aim to the precise identification of the cryptographic functions implemented.

Gröbert *et al.* [10] proposed in 2010 a work on cryptographic function identification. They define in particular several criteria to extract cryptographic parameters from execution traces and use I/O relationship comparison with known cryptographic functions. Parameters are identified thanks to the spatial proximity in the execution trace of instructions accessing their location in memory. In the presence of junk code, such a notion of proximity would be difficult to apply, because instructions responsible to access a same parameter can then be at a very variable distance. Moreover, candidate cryptographic parameters are extracted based on their size and therefore variable-length parameter functions like RC4 are hard to recognize. Zhao *et al.* also used I/O relationship to identify cryptographic functions [37]. Again, they made several assumptions on their programs, e.g. the ratio of exclusive-ors in cryptographic code or the use of certain types of functions, that are rarely satisfied in obfuscated programs.

Consequently, while their work opened the path to cryptographic identification based on I/O relationship comparison, it does not address identification within obfuscated programs. Implementing this identification is indeed far from trivial in such an environment. First, obfuscated programs lack abstractions that allow us to easily consider candidate parts of the code for identification. Second, obfuscation techniques tend to strongly increase the number of data accesses made by a program and therefore precise retrieval of cryptographic parameters becomes quite challenging.

3. SOLUTION OVERVIEW

A single I/O pair is enough to identify most cryptographic functions. We use this observation to identify them in obfuscated programs with the following three-step process:

1. **Gather execution trace of the targeted program.** Our identification technique needs the *exact* values manipulated by a program during an execution. Dynamic analysis is therefore particularly suitable and execution traces thus constitute our problem input. We will define execution traces formally in §4.

2. **Extract cryptographic code with I/O parameters from execution traces.** We use a specific definition of *loops* to build an abstraction suitable for cryptographic code detection in obfuscated programs. Secondly, we analyze data flow between loops in order to group those participating in the same cryptographic implementation, as there exists multi-loop cryptographic functions (e.g. RC4 [34]). We obtain a *loop data flow* model of the possible cryptographic code, from which we then extract I/O values. The complete building process from an execution trace to a set of loop data flows will be explained in §5 and 6.

3. **Comparison with known cryptographic functions.** Each extracted loop data flow is compared with a set of cryptographic reference implementations. If one of these implementations produces the same output than the loop data flow when executed on the same inputs, then we can conclude that they both implement the same cryptographic function. Despite its apparent simplicity, such comparison phase is non-trivial, because of the difference of abstraction between high-level reference implementations on one side, and execution traces on the other. This last step will be explained in §8.

4. EXECUTION TRACE GATHERING

In this work, we focus exclusively on the Windows/x86 platform. Among all tracing tools available in such environment, we chose Pin, the dynamic binary instrumentation framework supported by Intel [17], mainly because of its ease of use and its ability to deal with self-modifying code, a common technique in obfuscated programs. While we will not describe the tracer implementation, we do need to introduce a formal notion of execution traces as a basis for the rest of the reasoning about loops. An execution trace represents intuitively a series of operations done by a program during a run on a system. At each step, we collect what we call a *dynamic instruction*. A dynamic instruction D is a tuple composed of:

- a memory address $\mathcal{A}[D]$,
- a machine instruction $\mathcal{I}[D]$ executed at $\mathcal{A}[D]$,
- two sets of memory addresses read and written $\mathcal{R}_A[D]$ and $\mathcal{W}_A[D]$ by $\mathcal{I}[D]$,
- two sets of registers read and written $\mathcal{R}_R[D]$ and $\mathcal{W}_R[D]$ by $\mathcal{I}[D]$.

An *execution trace* T is a finite sequence $D_1; \ldots; D_n$ of dynamic instructions. In the remainder of the paper, we will denote as $\mathcal{X}86$ the set of machine instructions and as $TRACE$ the set of execution traces. Moreover, given $T \in TRACE$, $T_{/Ins}$ is defined as the machine instruction sequence, that is $T_{/Ins} = I_1; \ldots; I_n$ if $\forall k \in [1, n], \mathcal{I}[D_k] = I_k$. In practice we also gather the exact values for each data access made by a dynamic instruction (in memory or registers), but for the sake of simplicity we do not mention them in the formal trace notion.

5. LOOP EXTRACTION

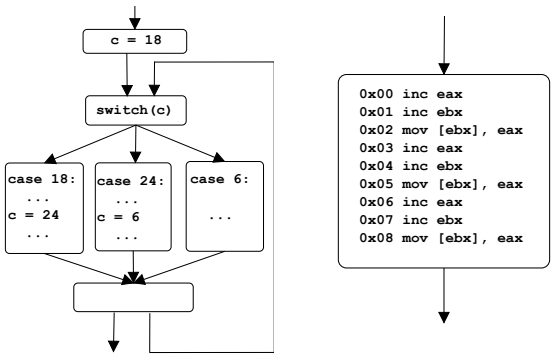
When analyzing binary programs, one can usually divide code into functions. But this function notion is only a heuristic based on compiler idiosyncrasies (calling convention, prologue and epilogue code, etc.), making it unreliable in obfuscated programs that do not maintain them. This is why we built a specific abstraction for cryptographic code in obfuscated programs based on loops. We will first motivate this choice, we will then present some existing loop definitions, and finally give our definition and describe the associated recognition algorithm.

5.1 Motivation and Related Work

As noted by Noe Lutz in his thesis, *"loops are a recurring feature in decryption algorithms"* [18]. Indeed cryptographic functions usually *apply the same treatment on a set of data in an iterative fashion*, making loops a very frequent structure in their implementations. It is within these loops that the core cryptographic operations happen and that the cryptographic I/O parameters are manipulated. Hence, they constitute a good starting point for our abstraction.

On the other hand, loops are present in many different types of algorithms, not only cryptographic ones. Thus, we need a refined loop notion focused on cryptographic code. For example Fig. 1(a) presents an obfuscation technique named *control-flow flattening* —in pseudo-C language for the sake of clarity— seen in the Mebroot malware family [31, 9]. A sequential code is transformed into a loop that implements a part of the original code at each iteration. Thus, a different logic is executed each time the back-edge is taken: should it be considered as a loop in our cryptographic context?

Secondly, Fig. 1(b) presents a classic compiler optimization technique called *unrolling* that can also be used as a means of obfuscation. A three-instruction sequence is repeated three times without any back-edge: should it be considered as a loop in our cryptographic context?



(a) Control-flow Flattening (b) Unrolling

Figure 1: Control-flow graph of loop edge cases

Several loop definitions for binary programs have been proposed in the literature. Three of them are of particular interest in the context of this work:

1. **Natural loops** This is the usual definition in static program analysis. Back-edges are defined on the program's control-flow graph (CFG) as edges between a node and one of its dominant [3]. Thus, a loop is identified by *one* back-edge, and an application of this definition on their CFG would consider Fig. 1(a) as a loop, but not Fig. 1(b).

2. **Address-centric loops** Another approach, especially built for dynamic analysis is that of [28]. A loop is identified by a specific memory address, named target address, where a *set* of backward branch instructions jump. In other words, several back-edges can correspond to the same loop, which is then identified by the target address. Nevertheless, this definition would also consider Fig. 1(a) as a loop, but not Fig. 1(b).

3. **Instruction-centric loops** Kobayashi defines loops on execution traces as the repetition of a machine instruction sequence [14]. For example, let T be a six instruction execution trace such that $T_{/Ins} = I_1; I_2; I_3; I_1; I_2; I_3$, then T is a loop iterating two times with $I_1; I_2; I_3$ as body. With this simple definition, Fig. 1(a) would not be considered as a loop, whereas Fig. 1(b) would be.

5.2 Simple Loop Definition

We focus on loops because cryptographic code usually use them to apply a *same treatment* on I/O parameters. Following this idea, Fig. 1(a) should *not* be considered as a loop —as a different logic is executed at each "iteration"— and on the contrary Fig. 1(b) should be. Therefore, we choose

the Kobayashi approach as our starting point: we identify a loop by a repeated machine instruction sequence, called its body. Thus, the body of the Fig. 1(b) loop is made of the three instructions `inc eax`, `inc ebx` and `mov [ebx],eax`.

A same loop can be run several times during an execution, each time with a different number of body repetitions. We call a particular run an *instance* of the loop. We also consider that the last iteration might be incomplete: a loop instance does not necessarily terminate at the exact end of its body.

We define this loop instance notion with formal language theory on the machine instruction alphabet $\mathcal{X}86$. For a word $\alpha \in \mathcal{X}86^*$ (i.e. α corresponds to a sequence of x86 instructions), we denote the set of prefixes of α by $Pref(\alpha)$, that is $\beta \in Pref(\alpha)$ if $\exists \gamma \in \mathcal{X}86^*, \alpha = \beta\gamma$. We denote as $\alpha \in \mathcal{X}86^+$ when $\alpha \in \mathcal{X}86^*$ and $|\alpha| \geq 1$.

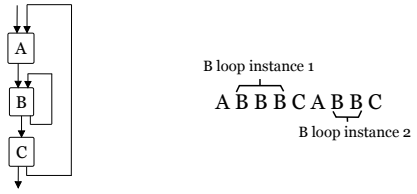
Definition 1. *The language SLOOP of simple loop instances is defined as all traces $\mathcal{L} \in TRACE$ such that:*

$$\mathcal{L}_{/Ins} \in \{\alpha^n.\beta | \alpha \in \mathcal{X}86^+, n \geq 2, \beta \in Pref(\alpha)\}$$

Thus, a simple loop instance is defined by at least two repetitions of a machine instruction sequence, called its body and represented by α. This definition is actually more general than Kobayashi's, who limited a loop body to contain the same machine instruction only once. In other words, there is no instruction $\ell \in \mathcal{X}86$ such that $\alpha = u\ell u'\ell u''$ in Kobayashi's work.

5.3 Nested Loop Definition

For our purposes, we also need to consider nested loops. Fig. 2(a) presents a common situation. Block B constitutes the body of an inner loop that does not iterate the same number of times for each outer loop iteration (cf. Fig. 2(b)). Consequently, if we directly apply Definition 1 on the execution trace, the outer loop would not be recognized as a loop. Nevertheless, it is still consistent with our loop principle: a same treatment applied repeatedly.



(a) CFG (b) Execution trace

Figure 2: Simplified nested loop example

It actually suffices to abstract each loop instance and apply Definition 1 recursively to solve this problem. Each time a loop instance is detected, we replace its code by a *loop identifier* in the execution trace. This identifier represents the loop associated with the instance. In other words, we replace each instance of a same loop with the same identifier. We introduce a set $\mathcal{L}_{ID}$ of loop identifiers, and we will use the letter X as a loop identifier in the rest of the paper. For example the execution trace in Fig. 2(b) can be rewritten as $AXCAXC$ with $X \in \mathcal{L}_{ID}$ the loop identifier for the inner loop B. The next application of Definition 1 will then be able to detect the outer loop with AXC as body.

We denote as $TRACE^{\mathcal{L}_{ID}}$ the set of execution traces where loop identifiers can replace dynamic instructions.

Definition 2. *The language LOOP of loop instances is defined as all traces $\mathcal{L} \in TRACE^{\mathcal{L}_{ID}}$ such that:*

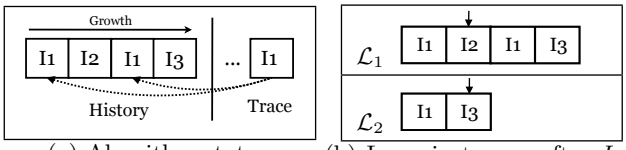
$$\mathcal{L}_{/Ins} \in \{\alpha^n.\beta | \alpha \in (\mathcal{X}86 \cup \mathcal{L}_{ID})^+, n \geq 2, \beta \in Pref(\alpha)\}$$

Let $\mathcal{L} \in LOOP$, we denote $BODY[\mathcal{L}] \in (\mathcal{X}86 \cup \mathcal{L}_{ID})^+$ the body of the loop instance $\mathcal{L}$, i.e. α in Definition 2.

5.4 Loop Instance Detection Algorithm

The loop detection algorithm is based on the recognition of the language $\{\alpha^n.\beta | \alpha \in (\mathcal{X}86 \cup \mathcal{L}_{ID})^+, n \geq 2, \beta \in Pref(\alpha)\}$, which is context-sensitive and in particular not context-free [13]. We built a *single pass* algorithm for loop instance detection, working in $O(m^2)$ operations, with m the execution trace size. We present the complete algorithm pseudo-code in Appendix B and, for the sake of brevity, we give here only an overview of how it works with an example.

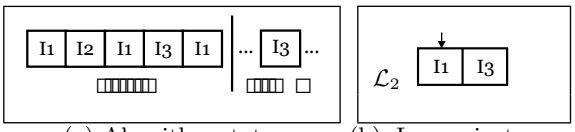
The *LOOP* recognition algorithm processes *machine instructions* from the execution trace one after the other, and stores them at the end of a list-like structure, named *history*. A common situation is then the one describes in Fig. 3(a): instructions I_1, I_2, I_1, I_3 have been recorded into the history and the currently processed machine instruction is I_1. Therefore this instruction appears twice in the history.



(a) Algorithm state (b) Loop instances after I_1 processing

Figure 3: Loop detection example: step one

Each occurrence of I_1 in the history corresponds to a possible loop instance beginning. In the first case, the body would be $\alpha = I_1; I_2; I_1; I_3$, whereas in the second one it would be $\alpha = I_1; I_3$. Thus, the algorithm creates two loop instances, named respectively $\mathcal{L}_1$ and $\mathcal{L}_2$, each of them with a *cursor* on the next instruction expected, I_2 for $\mathcal{L}_1$ and I_3 for $\mathcal{L}_2$ (cf. Fig. 3(b)).



(a) Algorithm state (b) Loop instances after I_3 processing

Figure 4: Loop detection example: step two

I_1 is then appended to the history. Now assume I_3 is the next machine instruction in the trace, the current situation is described in Fig. 4(a). $\mathcal{L}_1$ is then discarded, as it was not expecting this instruction. On the other hand, the $\mathcal{L}_2$ cursor is incremented and returns on the first instruction: it just made a turn (cf. Fig. 4(b)). At this point we have seen exactly two iterations for $\mathcal{L}_2$, that is $I_1; I_3; I_1; I_3$, and therefore we consider it as a *confirmed* loop instance. Thus, we replace its code in the history by its associated loop identifier $X \in \mathcal{L}_{ID}$, as described in Fig. 5.

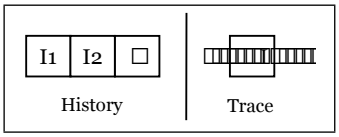


Figure 5: Loop detection example: step three

Suppose the next machine instruction is I_4. Then $\mathcal{L}_2$, which was waiting for I_1, is removed from running loop instances and registered. The replacement of its code with its loop identifier X will allow the detection of an outer loop, independently of the number of iterations of $\mathcal{L}_2$ for each outer loop iteration, as explained in §5.3.

6. LOOP INPUT-OUTPUT PARAMETERS

Loops allow possible cryptographic code extraction from execution traces, but our final objective is to collect cryptographic parameters. We present in this section a loop instance parameter notion and an algorithm to extract these parameters from execution traces. Then we will exhibit a simple example in order to make the understanding easier.

6.1 Definition

Loop instance parameters are low-level counterparts of high-level implementation parameters (called *high-level parameters* in the rest of the paper). Bytes read or written in the execution trace constitute our starting point and, for a loop instance $\mathcal{L}$, we define its parameters by combining the three following necessary conditions:

1. Bytes belonging to the same parameter of $\mathcal{L}$ are either *adjacent in memory*, or *in the same register at the same time*. This condition alone would tend to group multiple high-level parameters in the same parameter of $\mathcal{L}$. Indeed different high-level parameters can be adjacent in memory, as it is often the case in the stack. Such over-approximation would strongly complicate the final comparison phase; this is the reason we introduce the following condition.

2. Bytes belonging to the same parameter of $\mathcal{L}$ are manipulated *in the same manner* (read or written) by the *same* instruction in $BODY[\mathcal{L}]$. Indeed a particular instruction in $BODY[\mathcal{L}]$ can manipulate different bytes at each iteration but these data tend to have the same role (in particular because of our strict loop definition).

3. Finally, bytes belonging to an *input* parameter of $\mathcal{L}$ have been read without having been previously written by code within $\mathcal{L}$, whereas bytes belonging to an *output* parameter of $\mathcal{L}$ have been written by code in $\mathcal{L}$.

In order to gather these parameters, we define *concrete variables* as simple byte arrays starting at a particular memory address. If a concrete variable starts at the address `0x400000` and contains four bytes, we denote it as `0x400000:4`. Moreover, we also consider registers as concrete variables, whose addresses are register names, like `eax:4` or `bx:2`. The value contained in a concrete variable can change during the execution. A parameter is then defined as a concrete variable with a fixed value. For the sake of brevity, we now give only an overview of the parameter gathering algorithm.

```
; Prologue
LEA EAX,[KEY]
LEA EBX,[ENCRYPTEDTEXT]
PUSH SIZETODECRYPT
MOV ECX, 0                ; counter

; Core treatment
LOOP:
     MOV EDX,[EAX+ECX]    ; read 4 bytes from the key
     XOR [EBX+ECX], EDX   ; apply XOR operation
     ADD ECX,4            ; increment counter
     CMP ECX,[ESP]        ; is it finished ?
     JNZ LOOP

; Epilogue
ADD ESP,4                 ; rewind the stack
RETN
```

Figure 6: Assembly program implementing one-time pad cipher. `KEY`, `ENCRYPTEDTEXT` and `SIZETODECRYPT` are constants resolved by the assembler, the first two are memory addresses pointing to parameter values, whereas the last one contains their size.

6.2 Algorithm Overview

The algorithm groups bytes into concrete variables by using the first two necessary conditions. Then, we divide concrete variables into two groups, input and output parameters, by applying the third condition (the same concrete variable can be in both groups).

In a second step, the algorithm associates a fixed value to each previously defined concrete variables. As explained in §4, our tracer engine collects values for each data access. We use the two following rules to set parameter values: (1) the *first* time an input parameter is read provides its value, and (2) the *last* time an output parameter is written provides its value.

Finally, for each loop instance $\mathcal{L}$, the algorithm returns: $IN_M(\mathcal{L})$ and $IN_R(\mathcal{L})$ containing input parameters in memory and registers respectively, and $OUT_M(\mathcal{L})$ and $OUT_R(\mathcal{L})$ containing output parameters. The need for this distinction between memory and registers will become apparent in §7. The algorithm takes $O(m)$ steps, with m the execution trace size.

6.3 Example

In order to facilitate understanding of the previous definitions, we present here a simple artificial example. Fig. 6 presents an assembly language implementation of the one-time pad cipher [25], that is the application of a bitwise `XOR` operation between an input text and a key of the same length. We assembled this code into a program **P** that uses it with an 8-byte key `0xDEADBEEFDEADBEEF`, in order to decrypt the 8-byte text `0xCAFEBABECAFEBABE`. To identify the cryptographic function in **P** with the approach presented in this paper, we would need to collect the two input values along with the associated output result, namely `0x1453045114530451`. Therefore we gathered the execution trace of **P** with our tracer engine and applied the previous definitions on it. One loop instance was detected, whose instructions are presented in Table 1. Loop parameters were extracted by the previously described algorithm and we represent them in a *parameter graph* shown in Fig. 7.

Cryptographic parameters have been successfully retrieved:

Table 1: Loop instance code in the execution trace of P. Each set of adjacent bytes manipulated by an instruction is noted as a concrete variable.

D_i					
$\mathcal{A}[D_i]$	$\mathcal{I}[D_i]$	$\mathcal{R}_A[D_i]$	$\mathcal{W}_A[D_i]$	$\mathcal{R}_R[D_i]$	$\mathcal{W}_R[D_i]$
401011	mov edx, dword ptr [eax+ecx]	402000:4		eax:4 ecx:4	edx:4
401014	xor dword ptr [ebx+ecx], edx	402008:4	402008:4	ebx:4 ecx:4 edx:4	
401017	add ecx, 0x4			ecx:4	ecx:4
40101a	cmp ecx, dword ptr [esp]	12ffc0:4		esp:4 ecx:4	
40101d	jnz 0x401011				
401011	mov edx, dword ptr [eax+ecx]	402004:4		eax:4 ecx:4	edx:4
401014	xor dword ptr [ebx+ecx], edx	40200c:4	40200c:4	ebx:4 ecx:4 edx:4	
401017	add ecx, 0x4			ecx:4	ecx:4
40101a	cmp ecx, dword ptr [esp]	12ffc0:4		esp:4 ecx:4	
40101d	jnz 0x401011				

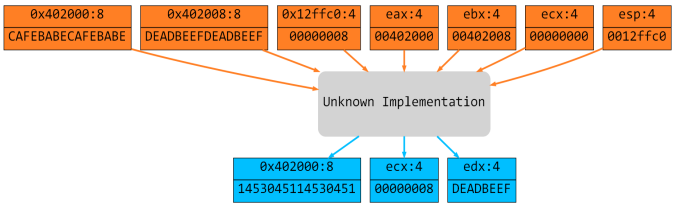


Figure 7: Parameter graph for P. Orange nodes are input-parameters, whereas blue ones are output parameters. Their values are noted under the line.

`402000:8` and `402008:8` as inputs, and `402000:8` as output. On the other hand, we also collected parameters with values linked to this particular implementation: (i) `eax:4`, `ebx:4`, `esp:4` containing memory addresses, (ii) `ecx:4`, containing a counter value, (iii) `12FFC0:4`, corresponding to a local variable initialized before the loop (`SIZETODECRYPT`), and (iv) `edx:4`, an intermediate storage. This will be taken into consideration in §8 for the comparison phase.

7. LOOP DATA FLOW

Until now we considered that each possible cryptographic implementation contains a single loop. Nevertheless, cryptographic functions can actually be composed of several non-nested loops, such as RC4 [34]. Consequently, the loop abstraction alone is not enough to capture them completely. In order to tackle this, we used data flow to group loop instances participating in the same cryptographic implementation. In this section we will describe the data flow construction algorithm leading to our final model and then we will define the associated parameters.

7.1 Loop Data Flow Construction

We define the data flow between loop instances in a similar way to *def-use chains* [3]: two loop instances $\mathcal{L}_1$ and $\mathcal{L}_2$ are connected if $\mathcal{L}_1$ produces an output parameter used by $\mathcal{L}_2$ as an input parameter. For the sake of simplicity, we consider only *memory* parameters, because register parameters would require a precise taint tracking inside sequential code between loop instances. Indeed, our assumption is that all treatments on inputs and outputs in memory are processed through loops.

Suppose that $\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$ is the set of loop instances extracted from $T \in TRACE$. We define a binary relation $\trianglelefteq$ between loop instances such that $\forall (i,j) \in [1,n]^2$, $\mathcal{L}_i \trianglelefteq \mathcal{L}_j$ if (i) $\mathcal{L}_i$ started before $\mathcal{L}_j$ in T, and (ii) $OUT_M(\mathcal{L}_i) \cap IN_M(\mathcal{L}_j) \neq \emptyset$. Next, we define the *loop data flow graph* $\mathcal{G}$ as $(\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}, \trianglelefteq)$. $\mathcal{G}$ is an acyclic graph, which may have several connected components $\mathcal{G}_1, \ldots, \mathcal{G}_m$, each of them with possibly several roots and leafs. For a connected component $\mathcal{G}_k$ we denote, respectively, by $ROOT[\mathcal{G}_k]$ and $LEAF[\mathcal{G}_k]$ the sets of root and leaf loop instances.

Each of these connected components represents an abstraction that is akin to functions in common binary programs. Thus, each $\mathcal{G}_k$ is a candidate cryptographic function implementation that will then be tested against known implementations. A standard graph algorithm is used to reconstruct the loop data flow graph, by testing the binary relation $\trianglelefteq$ for each pair of detected loops $\mathcal{L}_i$ and $\mathcal{L}_j$, and to detect its connected components. In the remainder of this paper, the connected components of a loop data flow graph will simply be called *loop data flows*.

In the case of composition between different cryptographic functions, that is, a function output used as input for another function, they will be grouped into the same loop data flow. A solution to this is to consider *every possible path* in the loop data flow graph. For example, suppose that $\mathcal{G}$ is the loop data flow graph $(\{\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3\}, \trianglelefteq)$, such that $\mathcal{L}_1 \trianglelefteq \mathcal{L}_2$ and $\mathcal{L}_2 \trianglelefteq \mathcal{L}_3$, then we test during the comparison phase not only the connected component $\{\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_3\}$, but also $\{\mathcal{L}_1, \mathcal{L}_2\}$, $\{\mathcal{L}_2, \mathcal{L}_3\}$, and finally each loop instance alone. Thus, we will be able to identify cryptographic functions used in combination with others.

7.2 Loop Data Flow Input-Output Parameters

Loop data flows constitute our model for cryptographic implementations and our final objective is the extraction of cryptographic parameters. We define loop data flow parameters as memory loop instance parameters *not used in the internal data flow*, i.e. in loop instance grouping. Regarding register parameters, we take input registers of the root loop instances and output registers of the leaf ones, for the sake of simplicity.

DEFINITION 3. *Let $\mathcal{G}_k = (\{\mathcal{L}_1, \ldots, \mathcal{L}_n\}, \trianglelefteq)$ be a loop data flow. Its* input parameters $IN_{\mathcal{G}_k}$ *are defined as*

$$\bigcup_{1 \leq i \leq n} \left(IN_M(\mathcal{L}_i) - \bigcup_{\mathcal{L}_j \trianglelefteq \mathcal{L}_i} OUT_M(\mathcal{L}_j) \right) \bigcup_{\mathcal{L}_r \in ROOT[\mathcal{G}_k]} IN_R(\mathcal{L}_r)$$

and its output parameters $OUT_{\mathcal{G}_k}$ *are defined as*

$$\bigcup_{1 \leq i \leq n} \left(OUT_M(\mathcal{L}_i) - \bigcup_{\mathcal{L}_i \trianglelefteq \mathcal{L}_j} IN_M(\mathcal{L}_j) \right) \bigcup_{\mathcal{L}_l \in LEAF[\mathcal{G}_k]} OUT_R(\mathcal{L}_l)$$

The *values* of these parameters have been collected during loop instance parameter extraction. Thus, we now have a model to extract possible cryptographic implementations from execution traces and collect their parameters. We can now identify cryptographic functions.

8. COMPARISON PHASE

The final step for our identification technique is the comparison of loop data flows with cryptographic reference implementations. We consider two different kinds of inputs:

- For each extracted loop data flow $\mathcal{G}_k$, we have the values of each parameter in $IN_{\mathcal{G}_k}$ and $OUT_{\mathcal{G}_k}$.
- For each common cryptographic function $\mathcal{F}$, we have a source code for a reference implementation $\mathbf{P}_{\mathcal{F}}$. In particular, a prototype describes its high-level parameters, e.g. whether they are of a fixed or variable-length.

The objective of the comparison algorithm is to check if the relationship between values in $IN_{\mathcal{G}_k}$ and $OUT_{\mathcal{G}_k}$ is also maintained by $\mathbf{P}_{\mathcal{F}}$. If so, it would imply that the code from which $\mathcal{G}_k$ was constructed implements $\mathcal{F}$. In other words, the actual output values of $\mathbf{P}_{\mathcal{F}}$ must match the output values in $OUT_{\mathcal{G}_k}$ when $\mathbf{P}_{\mathcal{F}}$ is executed on the input values from $IN_{\mathcal{G}_k}$. In this section we will first explain the hurdles we encountered during the comparison phase design, and then we will describe the actual comparison algorithm.

8.1 Difficulties

We use publicly available source codes as reference implementations. Consequently, there is a difference in the level of abstraction between the parameters extracted from execution traces and the ones defined in high-level source code.

Parameter type. Because what we extract with loop data flows are low-level parameters, i.e. contiguous memory addresses and registers, it is not necessarily obvious how to cast them back into the high-level types of the function reference implementation. Indeed, this could lead to artificial mismatches in the comparison, where the same value has been represented differently. This is why we choose reference implementations with parameters at the lowest possible level of abstraction, namely *raw values*.

Parameter order. High-level implementations declare their parameters in a specific order, but loop data flow parameters are not ordered. Consequently, we have to test all possible orders.

Parameter fragmentation. The same high-level parameter can be divided in several loop data flow parameters, e.g. when it is passed by registers but can not fit in only one of them. Therefore, loop data flow parameter values have to be combined to build the high-level parameter value. In other words, the mapping between loop data flow parameters and their high-level counterparts is not 1-to-1, but n-to-1.

Parameter number. Loop data flow parameters not only capture cryptographic parameters but also implementation-dependent ones, as shown in the §6.3 example. Thus, some loop data flow parameters will not have a matching high-level parameter.

8.2 Comparison Algorithm

The comparison algorithm tries to identify a loop data flow $\mathcal{G}_k$ with the following steps:

1. **Generation of all possible I/O values** Values in $IN_{\mathcal{G}_k}$ are combined by appending them to each other. We generate all combinations of all possible lengths. The same thing is done for output values in $OUT_{\mathcal{G}_k}$. For example, in §6.3 we generated as input parameters 5 values of length 4, 22 values of length 8 (all pairs of 4-byte values in both order, plus the two 8-byte values), etc. In practice the number of actual values generated can be strongly reduced if the analyst eliminates obvious implementation-dependent parameters, e.g. memory addresses.
2. **Input parameter mapping**. For each cryptographic reference implementation $\mathbf{P}_{\mathcal{F}}$, the algorithm selects for each high-level parameter its possible values among the ones generated in the previous step. In particular, for fixed-length parameters, only values with correct length are chosen.
3. **Comparison**. The program $\mathbf{P}_{\mathcal{F}}$ is executed on each possible combinations of its selected input values. If, at some point, the values produced are in the set of the possible output values generated in step 1, then it is a success. If this is not the case, the algorithm iterates until all combinations have been tested.

9. EVALUATION

We built a tool set, named *Aligot*, that implements the whole identification process, as described in Fig. 8. The tool is made of approximately 2000 lines of Python, plus 600 lines of C++ for the tracer engine. The set of cryptographic reference implementations has been built with the PyCrypto package [16].

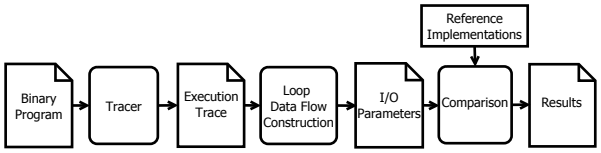


Figure 8: Aligot architecture

In order to evaluate Aligot, we used the following algorithms: the Tiny Encryption Algorithm (TEA) [35], the RC4 algorithm [34], the Advanced Encryption Standard (AES) [8] and the MD5 algorithm [23]. In addition, we explored the identification of modular multiplication in RSA [24]. For each of these algorithms, we compared Aligot with currently available cryptographic identification tools on both synthetic examples and malware samples.

Table 2: Detection results for various synthetic and malware samples. The name of the identified function is written and × indicates that no functions have been identified. Colors indicate the correctness of the result: green=correct, red=wrong.

	$\mathcal{B}_1$	$\mathcal{B}_2$	*Storm*	*SBank*	$\mathcal{B}_3$	*Sal* [a]	$\mathcal{B}_4$	$\mathcal{B}_5$	$\mathcal{B}_6$	$\mathcal{B}_7$	*Wal*	$\mathcal{B}_8$
Aligot	TEA	TEA	RTEA	RTEA	RC4	RC4	AES	AES	MD5	MD5	AES, MD5	MOD MUL
CryptoSearcher	TEA	×	×	×	×	×	AES	×	MD5	×	×	×
Draca	TEA	×	×	TEA	×	×	×	×	MD5	SHA-1	×	×
Findcrypt2	×	×	×	×	×	×	AES	×	MD4	×	×	×
H&C Detector	TEA	×	×	TEA	×	×	×	×	MD5	SHA-1	×	×
Kerckhoffs	×	×	×	×	×	×	×	×	×	×	×	×
PEiD KANAL	TEA	×	×	TEA	×	×	AES	×	MD5	×	×	×
Signsrch	TEA	×	×	TEA	×	×	AES	×	×	×	×	×
SnDCryptoS	×	×	×	×	×	×	AES	×	MD5	×	×	×

[a] Represents the four Sality samples Sal_1, Sal_2, Sal_3 and Sal_4

9.1 Tiny Encryption Algorithm

TEA is an 8-byte block cipher using a 16-byte key and is built as a 64-round Feistel network [35]. One of its particularities is the use of a constant value named *delta*, that is fixed at `0x9E3779B9` in its standard specification.

9.1.1 Detection Evaluation on Synthetic Examples

First, we verified the correctness of our identification process: we created 2 synthetic examples based on the source code published in the original TEA paper [35]. In each of these synthetic examples, we simply called once the TEA decryption function on the key `0xDEADBEE1...DEADBEE4` (16 bytes) and the encrypted text `0xCAFEBABECAFEBABE` (8 bytes). We then obtained the three following binary programs:

- $\mathcal{B}_1$: original TEA source code compiled with the Microsoft Visual Studio Compiler (MSVC) without any optimization (`/Od` option).
- $\mathcal{B}_2$: same as $\mathcal{B}_1$, except that we replaced the standard initialization of the *delta* value in assembly language (`mov reg, delta`) with an lightly obfuscated version using two instructions (`add reg, delta/2`). Consequently, the algorithm semantics is preserved but *delta* is no longer statically visible.

The detection results are presented in Table 2. We can interpret these results in the following manner:

- $\mathcal{B}_1$ allows to calibrate the evaluation. Tools failing to detect this classic implementation should not be considered as relevant, because they probably have no means to detect TEA. Nevertheless, we still mention their results for the sake of completeness.
- $\mathcal{B}_2$ is identified as a TEA implementation only by Aligot. We can then conclude that other tools base their detection solely on the static visibility of *delta*.

9.1.2 Detection Evaluation on Malware Samples

We investigated two malware families, called Storm Worm and SilentBanker, that were publicly referenced as using TEA [5, 22]. In both cases, the cryptographic implementation is part of the binary *protection layers*, that is, obfuscated code protecting the core logic. We collected one sample for each malware family, respectively *Storm* and *SBank*. The detection results are presented in Table 2.

Storm Worm. No TEA implementations were identified in the Storm Worm sample by Aligot. It means that, for each extracted loop data flow, its observed I/O relationship is not reproducible with a TEA reference implementation, according to our tool.

To confirm this, we isolated the part of the Storm Worm sample described as a TEA implementation by public sources [5] and analyzed it in-depth: (i) the code is a 8-byte block-cipher with a 16-byte key, exactly like TEA, (ii) the constant value *delta* is used during the decryption process, and (iii) arithmetic operations are similar to the ones used in TEA.

Nevertheless, we found a difference. Fig. 9(a) presents in pseudo-C a code line from the Storm Worm cryptographic implementation, whereas Fig. 9(b) shows the equivalent code line in the original TEA source code. In the malware version parenthesis are around exclusive-ors, whereas in the original code they are around additions. Hence the malware evaluates the expression in a different order than TEA and therefore produces different results.

$$z- = (y << 4) + (k[2] \oplus y) + (sum \oplus (y >> 5)) + k[3]$$

(a) Storm Worm

$$z- = ((y << 4) + k[2]) \oplus (y + sum) \oplus ((y >> 5) + k[3])$$

(b) TEA

Figure 9: Comparison between implementations

We believe malware authors simply made a "copy/paste" from existing code containing this error. In fact, we found the same assembly code on a Russian website [30]. Therefore, we named this new cryptographic function "Russian-TEA". We confirmed this observation by creating a reference implementation for the Russian-TEA function. Aligot then successfully identified the code in Storm Worm as a Russian-TEA implementation, as indicated in Table 2.

Table 4: Comparison between RC4 implementations identified by Aligot, where NBB is the number of Basic Blocks and IPBB is the average number of instructions per Basic Block.

	$\mathcal{B}_3$	Sal_1	Sal_2	Sal_3	Sal_4
NBB	18	9	17	4	14
IPBB	7	40	25	97	29

Silent Banker. Strangely enough, Aligot found no TEA implementations in the SilentBanker sample either. However, Aligot automatically identified a Russian-TEA implementation in it, something that we completely ignored before running Aligot. After manual investigation, we confirmed that indeed the two malware families made the exact same mistake. As a side note, SilentBanker was believed to have been created in Russia [29]. Moreover, other tools identified SilentBanker as containing a TEA implementation, because the delta constant is statically visible in this case.

In summary, Aligot successfully identified that, despite appearances, Storm Worm was *not* implementing TEA. Moreover, after adding a reference implementation for this new cryptographic function (Russian-TEA), Aligot automatically identified the same function in SilentBanker, whereas other tools wrongly identified it as TEA.

9.2 RC4

The RC4 algorithm is a stream cipher using a variable-length key [34]. A pseudo-random stream of bits is generated with the key, then an XOR operation is done with the input text in order to decrypt or encrypt it.

9.2.1 Detection Evaluation on Synthetic Example

Similarly as for TEA, we first verified the correctness of our identification process. We created a synthetic example by compiling a reference RC4 source code [34] with MSVC without any optimization, that we denote as $\mathcal{B}_3$ in Table 2.

The detection result is presented in Table 2. Aligot is the only tool able to identify this binary as RC4 implementation. Indeed this cryptographic function lacks particular static signs (such as fixed values), making it quite difficult to recognize with classic tools.

9.2.2 Detection Evaluation on Malware Samples

We investigated a malware family, named Sality, that was publicly referenced as using RC4 in its protection layers [36]. We collected four Sality samples on the Internet: Sal_1, Sal_2, Sal_3 and Sal_4. The detection results are presented in Table 2 in the column Sal, as they are the same for the four samples. Again, Aligot is the only tool able to identify an RC4 implementation in all binary programs.

As with the previous malware samples, the cryptographic implementation is part of the protection layers and is heavily obfuscated. Moreover, Sality samples contain a variety of obfuscation techniques. In order to demonstrate this, we provide in Table 4 a structural comparison between RC4 implementations in all binary programs, based on the number of *basic blocks* (BB) of the CFG and the average number of instructions per BB (this comparison concerns only parts of each program recognized as RC4 implementations by Aligot).

Moreover, thanks to the parameter graphs extracted from each Sality RC4 implementation, we were able to identify a pattern in the way the malware uses its cryptographic parameters (two of these graphs are presented in Appendix A). Each cryptographic parameter (key, input and output text) is at the same offset from the beginning of the file in each sample, even though their values are different. This information is valuable to build a static unpacker for Sality, as it gives a generic path to access cryptographic parameters.

9.3 AES

The AES algorithm is a 16-byte block cipher with a key of either 128, 256 or 512 bits [8]. It processes input data through a substitution-permutation network where each iteration (round) employs a round key derived from the input key. Unlike in the simpler TEA and RC4 algorithms, the first and the last of these rounds are done outside the main loop of AES, as they are functionally different than the other inner rounds. Consequently, when a comparison with an AES reference implementation is made with Aligot, only the I/O relationship for these inner rounds will be verified, not for the parameters of the AES algorithm in its entirety.

9.3.1 Detection Evaluation on Synthetic Example

We compiled a reference source code with the OpenSSL library [12] without any optimization, where we simply encrypt a particular input text with AES-128, giving us the program $\mathcal{B}_4$. As described in Table 2, most of the tools, including Aligot, are able to identify such a program. Nevertheless, Aligot is the only one able to infer the actual key size, thanks to its knowledge of the parameters (in this case the round keys).

9.3.2 Detection Evaluation on Obfuscated Programs

To show the resistance of our tool against obfuscation techniques, we tested it on the two following binary programs: 1) the program $\mathcal{B}_5$, which is the result of the application of the commercial packer AsProtect [1] on $\mathcal{B}_4$, and 2) Wal, which is a sample of the Waledac malware family, which was already known to use AES for encrypting its Command and Control traffic [7].

In both cases, Aligot is the only tool to successfully identify AES-128.

9.4 MD5

The MD5 algorithm is a cryptographic hash function that produces a 128-bit hash value [23]. The input message is broken up into chunks of 512-bit that are then processed in an iterative fashion. This core iterative behavior offers an opportunity for Aligot to identify this algorithm.

9.4.1 Detection Evaluation on Synthetic Example

Similarly as for AES, we compiled a reference source code from the OpenSSL library [12] without any optimization, where we simply hash a particular input text, giving us the program $\mathcal{B}_6$. As presented in Table 2, most of the tools, including Aligot, are successful in identifying MD5.

9.4.2 Detection Evaluation on Obfuscated Programs

The obfuscated test cases were: 1) the program $\mathcal{B}_7$, produced using AsProtect on $\mathcal{B}_6$, and 2) Wal, the same sample of the Waledac family that uses MD5 in its payload to compute unique ID for each bot.

Again, Aligot is the only tool to successfully identify MD5, as described in Table 2.

Table 3: Aligot performance. LDF stands for "Loop Data Flow", DI for "Dynamic Instruction"

	$StormW$	$SilentB$	Sal_1	Sal_2	Sal_3	Sal_4	Wal
Trace size (DI)	3M	3.5M	4.1M	1M	4.8M	4.2M	20k
Tracing (min)	4	3	2	1	2	2	1
LDF construction	4hr	6hr	10hr	4hr	10hr	15hr	40mn
Comparison (min)	30	30	3	3	4	4	10
Total time	4,5hr	6,5hr	10hr	4hr	10hr	15hr	51mn

9.5 RSA

The RSA algorithm [24] is an asymmetric cipher whose security is based on the hardness of factoring large integers. The two operations used for encryption and decryption are modular exponentiation and modular multiplication. Modular exponentiation is typically implemented as a loop where each iteration consists of either one multiplication or one squaring and a multiplication, depending on the bit of the exponent being referenced. This creates an outer loop pattern that is not directly detectable with Aligot because the trace generated does not belong to the $LOOP$ language of Definition 2. Nonetheless, the inner modular multiplication loops are detectable and can serve as the basis for detection of RSA or similar algorithms.

We built a synthetic example of an RSA encryption, denoted as $\mathcal{B}_8$, using the PolarSSL library [2] and also protected it with AsProtect. Since this library, as most RSA implementations, uses the optimized Montgomery algorithm [20] for modular multiplication, we added a reference implementation of this algorithm to the database, i.e. a sample non-obfuscated program performing Montgomery multiplication also constructed using Polar SSL. One of the problems with RSA (and of other public-key algorithms) is the multiplicity of possible parameter encoding (in this case large integers). Since Aligot retrieves parameters in the execution trace in their low-level encoded form (i.e. representation in memory), this might not match the high-level representation of I/O parameters of the reference implementations (programmed in high-level languages). To deal with this, care has to be taken when constructing the reference implementations that the same parameter representation is used as in the compiled machine code.

With these precautions, Aligot successfully identifies the presence of modular multiplication operation (denoted as "MOD MULT" in Table 2) in the obfuscated code of $\mathcal{B}_8$, while none of the other tools are able to do so.

9.6 Further Results

Since compiler parametrization can significantly changed the structure of compiled code, we decided to test Aligot and the other tools with samples compiled using various space and speed optimization options. The performance of Aligot was unaffected by these options, while some of the other tools did lose their ability to identify the corresponding crypto algorithm.

Furthermore, we considered the possibility that certain implementations of crypto algorithms using fixed constants, might use different values of these constants (without affecting their security) in order to avoid detection. In order to explore this possibility, we tested Aligot against TEA implementations with modified *delta* constant values. Thanks to the use of a parametrized reference implementation TEA, where the *delta* constant becomes an input parameter, Aligot was still able to detect these modified TEA implementations. All other tools failed to do so, which seems to confirm that their detection of TEA is based solely on recognition of the standard *delta* constant value.

Finally, and as discussed at the end of §7.1, we also tested Aligot against sample code where several crypto functions were combined, e.g. Double TEA (two TEA applications in a row), or a combination of one-time pad and RC4. As expected, this was successfully detected by Aligot.

9.7 Performance

Table 3 presents our performance results for detection on malware samples on a common end-user computer; the performance for synthetic samples is comparatively much quicker (in all cases under 20 min) and not shown. These results should be considered as purely indicative, as Aligot was built in first place to prove the feasibility of the approach, and not to be a particularly efficient tool. As can be seen in Table 3 larger trace sizes do not always translate into longer processing times (e.g Sality samples Sal_3 and Sal_4). Indeed, the longest task is loop detection, and while we gave a worst-case complexity estimate for it as a function of trace size in §5.4, in practice its performance also strongly depends on the number of loops. If loops are rare (according to our definition), then history size grows and performance decreases. Regarding the comparison phase, its performance depends in practice on the number of loop data flows extracted and their actual number of parameters.

10. LIMITATIONS

First of all, and like any other dynamic analysis tool, Aligot is only able to identify code that is actually executed at run time. Identifying which potential paths are relevant for analysis and should be "fed" to such a tool is a separate research problem. Nonetheless, in practice and for most malware samples, this can be handled with success by a skillful malware analyst.

Second, Aligot's identification ability is limited to those functions for which we possess a reference implementation. Nevertheless, the design of the tool allows the analyst to easily add new functions to the database, as it was demonstrated for RTEA (as discussed in §9.1).

Despite the experimental results presented in this paper, we are not claiming that our loop data flow model captures *every* possible cryptographic function obfuscated implementation. Indeed, computer program analysis is inherently undecidable, as stated by Rice's theorem [13], and this theoretical impossibility is particularly relevant in the adversarial game of obfuscation. In our case, malware authors could simply implement cryptographic functions with loops that do not correspond to our definition. In response to this

move, we could define and use a more inclusive loop model. Another counter-measure for malware writers could be to encode their parameters in such a way it does not correspond to standard encodings used by our high-level reference implementations. Such a technique could be bypassed by adding specific decoding procedures in front of our reference functions, as exhibited in §9.5. While accepting that no perfect solution exists, a clear definition of the analysis context can lead to answers for specific cases.

11. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a method for cryptographic function identification and parameter extraction in obfuscated programs. To this end, we introduced loop data flows as a substitute for higher-level abstractions in obfuscated code. Based on this abstraction, we extracted input and output parameters and compared them with those of known cryptographic functions. We built the Aligot tool and tested it against synthetic obfuscated programs and actual malware samples containing known cryptographic functions, such as the Tiny Encryption Algorithm (TEA), RC4, MD5, AES and RSA.

The main result is that Aligot performs significantly better than all previous tools, which is not surprising since they were not really designed for analysis of obfuscated code. It is important to underline that while most other tools are defeated by simple techniques, like changes in compiler options and hiding of easily recognizable features, Aligot's ability remains the same. Aligot is also the only tool that is able to detect these crypto functions on samples packed with a commercial-grade code protector (AsProtect) and in mildly obfuscated malware such as Storm, Silent Banker and Waledac. Furthermore, Aligot is even able to perform adequately in the presence of important changes in code structure due to severe obfuscation, such as the case of the Sality malware.

Most importantly, even though we experimented with a relatively small set of samples and cryptographic functions, we were able to find encouraging evidence of the usefulness of cryptography identification in obfuscated malware. In the case of Sality, for example, using Aligot we were able to identify an invariant feature in each of the four samples examined (fixed offset of cryptographic parameters) that could be used to automatically access the protected code (i.e. used by static unpackers). Most surprisingly, we were able to detect that the Storm Worm and Silent Banker binaries shared the same error in the implementation of the TEA cryptographic function. This is a strong indicator that malware authors were either the same or were using the same code base. This supports our proposition that analysis of cryptographic code in malware could provide important information leading up to attribution of malware activities to particular groups or individuals.

One of the limitations of our work, is the inability of the current loop model to detect a full implementation of RSA, due to the conditional behavior of the modular exponentiation loop. While it is possible to modify the loop language to include such conditional iterations, care has to be taken to avoid including "artificial" loop constructions, such as those generated by the control-flow flattening techniques discussed in §5. Given the relatively simple conditional structure of RSA (only two possible choices), it would be possible to slightly modify to the loop language to include it, while excluding undesirable loop-like code.

As mentioned in §10, Aligot is only able to recognize crypto that is known and for which a reference implementation has been constructed. In principle, this would prevent the detection of "home-made" or other unknown crypto used by malware authors in their code. However, our experience with Russian TEA is inspiring: with some manual work we were able to detect an unknown crypto algorithm (albeit, an incorrect implementation of a known one). This suggests an automated approach by which an unknown algorithm could be detected, as long as an implementation for it has been included into the reference database. The idea would be to create an "import" tool able to extract reference implementations from unknown obfuscated binary samples, for example from large databases of captured malware. By using the loop detection algorithm of Aligot, this tool could reconstruct the binary code implementing the loops of interest (possibly implementing crypto functions) that could then be included into a reference implementation database. A streamlined version of Aligot could then be used to find possible matches in this large database. Since the detection process presented in this paper can be extended to any type of functions maintaining a particular I/O relationship (e.g. compression functions), this could provide the basis for a new and promising methods of malware classification.

Availability. To encourage follow-up research, we have made available (1) the complete code for Aligot, (2) the set of binaries used for experimental evaluation in this paper, and (3) the version information and links for the other tools at `http://code.google.com/p/aligot/`.

12. ACKNOWLEDGEMENTS

We would like to thank Pierre-Marc Bureau, Pierre Brun-Murol and Nicolas Falliere for their many helpful comments during this research project.

13. REFERENCES

[1] Asprotect packer. `http://www.aspack.com/asprotect.html`.

[2] Polar SSL library Web site. `http://polarssl.org`.

[3] A. Aho, J. Ullman, and S. Biswas. *Principles of Compiler Design*. Addison-Wesley, 1977.

[4] L. Auriemma. Signsrch tool. `http://aluigi.altervista.org/mytoolz.htm`.

[5] F. Boldewin. Peacomm.c Cracking the nutshell. `http://www.reconstructer.org/papers/Peacomm.C-Crackingthenutshell.zip`.

[6] J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proc. 16th ACM Conf. on Computer and Communications Security (CCS)*, pages 621–634, 2009.

[7] J. Calvet, C. Davis, and P. Bureau. Malware authors don't learn, and that's good! In *Proc. 4th Int. Conf. on Malicious and Unwanted Software (MALWARE)*, pages 88–97. IEEE, 2009.

[8] J. Daemen and V. Rijmen. *The design of Rijndael: AES–the advanced encryption standard*. Springer-Verlag, 2002.

[9] N. Fallière. Reversing Trojan.Mebroot's Obfuscation. In *Reverse Engineering Conference (REcon)*, 2010.

[10] F. Gröbert, C. Willems, and T. Holz. Automated identification of cryptographic primitives in binary programs. In *Proc. Recent Advances in Intrusion Detection (RAID)*, pages 41–60. Springer, 2011.

[11] J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten. Lest we remember: cold-boot attacks on encryption keys. *Comm. of the ACM*, 52(5):91–98, 2009.

[12] S. Henson et al. OpenSSL library. `http://openssl.org`.

[13] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, 2007.

[14] M. Kobayashi. Dynamic characteristics of loops. *IEEE Trans. on Computers*, 100(2):125–132, 1984.

[15] I. O. Levin. Draft crypto analyzer (draca). `http://www.literatecode.com/draca`.

[16] D. Litzenberger. PyCrypto - The python cryptography toolkit, 2011.

[17] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices*, 40:190–200, 2005.

[18] N. Lutz. Towards revealing attacker's intent by automatically decrypting network traffic. Master's thesis, ETH Zürich, Switzerland, 2008.

[19] C. Maartmann-Moe, S. Thorkildsen, and A. Arnes. The persistence of memory: Forensic identification and extraction of cryptographic keys. *Digital Investigation*, 6:S132–S140, 2009.

[20] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, 1985.

[21] M. Morgenstern and H. Pilz. Useful and useless statistics about viruses and anti-virus programs. In *Proc. CARO Workshop*, 2010.

[22] L. O Murchu. Trojan.silentbanker decryption. `http://www.symantec.com/connect/blogs/trojansilentbanker-decryption`.

[23] R. Rivest. RFC 1321: The MD5 message-digest algorithm. *Internet Activities Board*, 143, 1992.

[24] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. of the ACM*, 21(2):120–126, 1978.

[25] C. E. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4):656–715, 1949.

[26] N. Stewart. Inside the storm: Protocols and encryption of the Storm botnet. In *Black Hat Technical Security Conference*, 2008.

[27] S. Trilling. Project Green Bay - Calling a Blitz on Packers. In *CIO Digest: Strategies and Analysis from Symantec*, 2008.

[28] J. Tubella and A. González. Control speculation in multithreaded processors through dynamic loop detection. In *Proc. 4th Int. Symp. on High-Performance Computer Architecture*, pages 14–23. IEEE, 1998.

[29] VeriSign. Silentbanker analysis. `http://www.verisign.com/static/043671.pdf`.

[30] Russian TEA assembly code. `http://www.xakep.ru/post/22086/default.asp`.

[31] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, 2000.

[32] Z. Wang, X. Jiang, W. Cui, X. Wang, and M. Grace. ReFormat: Automatic reverse engineering of encrypted messages. In *Proc. ESORICS*, pages 200–215, 2009.

[33] PEiD Krypto Analyzer (kanal). `http://www.peid.info`.

[34] RC4 source code. `http://cypherpunks.venona.com/date/1994/09/msg00304.html`.

[35] D. Wheeler and R. Needham. TEA, a tiny encryption algorithm. In *Proc. Fast Software Encryption*, pages 363–366. Springer, 1995.

[36] V. Zakorzhevsky. A new version of Sality at large. `http://www.securelist.com/en/blog/180/A_new_version_of_Sality_at_large`.

[37] R. Zhao, D. Gu, J. Li, and R. Yu. Detection and analysis of cryptographic data inside software. *Information Security*, pages 182–196, 2011.

APPENDIX

A. LOOP INSTANCE DETECTION ALGORITHM

We used the following data structures to implement the algorithm:

- *History*: list-like structure storing machine instructions already seen along with loop IDs. In the pseudo-code the $History$ variable is named H.
- *LoopInstance*: structure containing information related to a loop instance.
- *RunningLoops*: set of running loops; each of these items is a stack representing a loop nesting: the head contains the *LoopInstance* of the most nested loop, whereas the rest of the stack contains the higher loops. In the pseudo-code the *RunningLoops* variable is named RL.

The main procedure pseudo-code is given in Algorithm 1, whereas the core logic of the loop detection is contained in the recursive procedure $Match()$ given in Algorithm 2. Some operations are described directly in natural language to ease the understanding and some details are hidden for the same reason.

The algorithm is build on a simple rule: a machine instruction is either part of a confirmed loop instance or a possible beginning for such instances, but *not both at the same time*. This dichotomy is expressed in the main procedure: for each instruction the for-loop checks if a confirmed instance waits for it and if so — *Match()* procedure returns 1 — the others instances are not tested and the instruction is not considered as a possible loop beginning. On the other hand, if no confirmed loop instance waits for the instruction, then the second part of the main procedure checks if it could be a

loop beginning. During this process the history stores machine instructions that have not been (yet) proved to be part of a confirmed loop.

Algorithm 1 Loop Instance Detection Main Procedure

Require: T : $TRACE_{x86}$, H : $History$, RL : $RunningLoops$

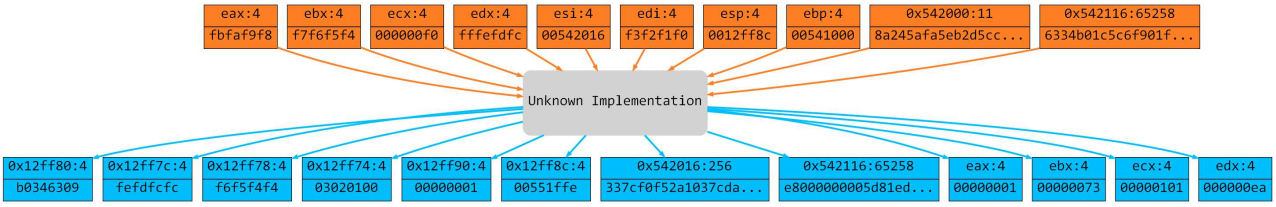
for $i = 1$ to $Length(T)$ **do**
  $ConfirmedInstanceWaitsForMe \leftarrow 0$
  for $StackOfLoops$ in RL **do**
    if Match($StackOfLoops$, $\mathcal{I}[D_i]$, $History$) $= 1$ **then**
      $ConfirmedInstanceWaitsForMe \leftarrow 1$
      Break
    end if
  end for
  if $ConfirmedInstanceWaitsForMe = 0$ **then**
    Append(H, $\mathcal{I}[D_i]$)
    if there exists others $\mathcal{I}[D_i]$ occurrences in H **then**
      Create associated loop instances
      Add them to RL
    end if
  end if
end for

B. PARAMETER GRAPHS

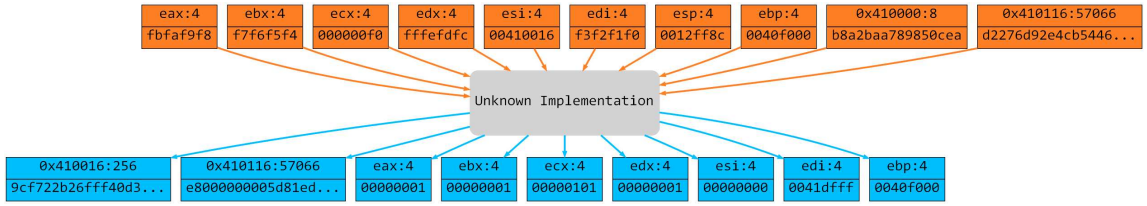
Algorithm 2 Loop Instance Detection Match Procedure

Require: $StackOfLoops$: $Stack(LoopInstance)$, I_j : $\mathcal{X}86$, H : $History$

if $StackOfLoops$ is empty **then**
  return 0 {Base case: failure}
end if
$currentInstance \leftarrow$ Head($StackOfLoops$) {Get the most nested loop}
if $currentInstance.cursor$ points to a loop ID X **then**
  Increment $currentInstance.cursor$
  Create new instance I for the loop X
  Push($StackOfLoops$, I)
  return Match($StackOfLoops$,I_j,H)
else
  if $currentInstance.cursor$ points to I_j **then**
    Increment $currentInstance.cursor$
    if $currentInstance$ iterates for the second time **then**
      Remove $currentInstance$ instructions from H
      $currentInstance.confirmed \leftarrow 1$
      Append(H, $currentInstance.ID$)
    end if
    if $currentInstance.confirmed = 1$ **then**
      return 1 {Success}
    else
      return 0 {Failure}
    end if
  else
    if $currentInstance.confirmed = 1$ **then**
      Pop($StackOfLoops$)
      return Match($StackOfLoops$,I_j,H)
    else
      Discard $StackOfLoops$
      return 0 {Failure}
    end if
  end if
end if



(a) Sal_1 parameter graph for the part recognized as RC4 by Aligot: `542000:11` is the key, `542116:65258` the input text and `542116:65258` the output text.



(b) Sal_2 parameter graph for the part recognized as RC4 by Aligot: `410000:8` is the key, `410116:57066` the input text and `410116:57066` is the output text.

Figure 10: Parameter graphs. Only the first 8 bytes of parameter values are displayed. Every cryptographic parameter starts at the same offset from the file beginning (`0x542000` for Sal_1 and `0x410000` for Sal_2).